¿Por qué hacer testing unitario?

Esta es una buena pregunta que nos podemos hacer todos los implicados en un proyecto de **desarrollo de software** y en especial los desarrolladores. Debemos ser los primeros interesados en hacer test unitarios y no sólo por el hecho de hacerlos y ser un indicativo más de cara a la galería. Se deben hacer test unitarios de igual forma que si se tratase código de producción.

El desarrollo de software, al igual que otros productos que utilizamos diariamente, debe ser testeado. En la cocina por ejemplo, ¿os imagináis que los platos que nos sirven no los pruebe antes el chef?; en automoción todas las piezas de un coche deben pasar pruebas de calidad; o cuando nos compramos un coche, esperamos que todo funcione perfectamente; y así innumerables ejemplos. En todos los sectores las pruebas tienen un papel muy importante. Y, por tanto, no puede ser menos importante en nuestro sector. Nos enfrentamos a usuarios de distintas generaciones, múltiples dispositivos y, si estamos en el mundo web, múltiples exploradores. Además los usuarios queremos aplicaciones rápidas, seguras, usables, etc.

Una vez que el equipo de desarrollo de software asume el hacer test como una parte más de su trabajo, es muy importante ser mentor al resto de intervinientes en el proyecto.

Un sistema con test unitarios será más fácil cambiarlo con la seguridad de que no vamos a romper nada. Nos permitirá realizar tareas de refactoring sin miedo y buscar posibles bugs de forma más rápida. Además al hacer test, se detectarán malas prácticas de diseño y asumimos estas mejoras en etapas tempranas del desarrollo, lo que implica una mejor calidad del sistema.

Experiencias sin testing

Aunque puede parecer extraño no hace muchos años, e incluso en la actualidad, hay proyectos de software que se desarrollan y se suben a producción sin ningún test. Bueno, esto no es cierto del todo, siempre hay algún responsable del proyecto que realiza “sus pruebas” y da el OK al software. Está bien realizar este tipo de pruebas pero no te aseguran que el software es robusto.

En muchas ocasiones se ha arreglado el software por una incidencia y a la vez se han generado muchas más por otro sitio. ¿Cómo evitar estos derroches de trabajo y horas sin disfrutar con tus amigos o familia?

Todos debemos ser conscientes (desde el programador junior hasta los gerentes de grandes cuentas) de la importancia de realizar testing. ¿Qué tipo de testing? Cuanto más, mejor, pero siempre con sentido común y **no hacer test simplemente para mejorar los números del Sonar**de cara a la galería.

Si eres desarrollador Java seguro que estas excepciones las has vistos en algún log de producción:

java.lang.NullPointerException, java.lang.ArrayIndexOutOfBoundsException

Quizás con unos buenos **test unitarios** se hubieran podido evitar.

Al igual que éstas, que son más complejas de analizar:

- java.lang.OutOfMemoryError

Con unas buenas **pruebas de carga** también se hubieran podido detectar.

- java.lang.UnsupportedClassVersionError

Este tipo de excepciones son más difíciles de detectar con testing ya que dependen de la versión de la JVM/librerías. Para asegurarse de detectarlas a tiempo es importante que la infraestructura, arquitectura y JVM dispongan de las mismas versiones en todos los entornos.

Pero como dice Edsger Dijkstra:

"El testing puede probar la presencia de errores pero no la ausencia de ellos"

A parte de test unitarios, ¿qué otros tipos de test existen?

Ya hemos mencionado algunos tipos de test: unitarios y de carga. Hay varios tipos de test, incluso se podrían añadir alguno más que vuestros sistema requiera:

- **Test unitarios**: prueban una funcionalidad única y se basan en el principio de responsabilidad única (la S de los principios de diseño SOLID)
- **Integración**: prueban la conexión entre componentes, sería el siguiente paso a los test unitarios.
- **Funcionales (o Sistema)**: prueban la integración de todos los componentes que desarrollan una funcionalidad concreta (por ejemplo, la automatización de pruebas con Selenium serían test funcionales).
- **Aceptación de Usuarios**: Pruebas definidas por el Product Owner basadas en ejemplos (BDD con Cucumber). Para más detalle [Link a entrada blog cucumber].
- **Regresión**: Prueban que los test unitarios y funcionales siguen funcionando a lo largo del tiempo (se pueden lanzar tanto de forma manual como en sistemas de Integración Continua).
- **Carga**: Prueban la eficiencia del código.

Una inmersión en los tests unitarios

Los tests unitarios prueban las funcionalidades implementadas en el SUT (System Under Test). Si somos desarrolladores Java, para nosotros el SUT será la clase Java.

Los tests unitarios deben cumplir las siguientes características:

Principio FIRST

- **Fast**: Rápida ejecución.
- **Isolated**: Independiente de otros test.
- **Repeatable**: Se puede repetir en el tiempo.
- **Self-Validating**: Cada test debe poder validar si es correcto o no a sí mismo.
- **Timely**: ¿Cuándo se deben desarrollar los test? ¿Antes o después de que esté todo implementado? Sabemos que cuesta hacer primero los test y después la implementación (TDD: Test-driven development), pero es lo suyo para centrarnos en lo que realmente se desea implementar.

Además podemos añadir estos dos puntos más:

- Sólo pruebas los **métodos públicos** de cada clase.
- **No** se debe hacer uso de las **dependencias** de la clase a probar. Esto quizás es discutible porque en algunos casos donde la dependencias son clases de utilidades y se puede ser menos estricto. Se recomienda siempre aplicar el sentido común.
- Un test **no debe implementar ninguna lógica de negocio** (nada de if...else...for...etc)

Los tests unitarios tienen la siguiente estructura:

- Preparación de datos de entrada.
- Ejecución del test.
- Comprobación del test (assert). No debe haber más de 1 assert en cada test.

Framework Java testing: JUnit 4

JUnit es un framework Java para implementar test en Java. Se basa en anotaciones:

- **@Test**: indica que el método que la contiene es un test: expected y timeout.
- **@Before**: ejecuta el método que la contiene justo antes de cada test.
- **@After**: ejecuta el método que la contiene justo después de cada test.
- **@BeforeClass**: ejecuta el método (estático) que la contiene justo antes del **primer test**.
- **@AfterClass**: ejecuta el método (estático) que la contiene justo después del **último test**.
- **@Ignore**: evita la ejecución del tests. No es muy recomendable su uso porque puede ocultar test fallidos. Si dudamos si el test debe estar o no, quizás borrarlo es la mejor de las decisiones.

Las condiciones de aceptación del test se implementa con los asserts. Los más comunes son los siguientes:

- **assertTrue/assertFalse** (condición a testear): Comprueba que la condición es cierta o falsa.
- **assertEquals/assertNotEquals** (valor esperado, valor obtenido). Es importante el orden de los valores esperado y obtenido.
- **assertNull/assertNotNull (object)**: Comprueba que el objeto obtenido es nulo o no.

- **assertSame/assertNotSame(object1, object2)**: Comprueba si dos objetos son iguales o no.
- **fail()**: Fuerza que el test termine con fallo. Se puede indicar un mensaje.

Además existen otras posibilidades más avanzadas:

- **Suites**: es una colección de un conjunto de test que se ejecutan de forma independiente (@RunWith(Suite.class),@SuiteClasses({})
- **Runners**: Definen cómo ejecutar los test (@RuntWith)
- **Test parametrizados**: test genéricos que se ejecutan con juegos de datos distintos (@Parameters)
- **Rules**: Pueden ejecutar código antes, después o dentro de los métodos (@Rule)

En este último año ha salido la versión 5 de JUnit. En el siguiente enlace podéis encontrar las nuevas características. Mencionar que es compatible con las versiones anteriores ya que ha mantenido la paquetería org.junit para la implementación anterior y para JUnit 5 org.junit.jupiter.

Mockito y los dobles

Mockito es una librería Java que permite simular el comportamiento de una clase de forma dinámica. De esta forma nos aislamos de las dependencias con otras clases y sólo testeamos la funcionalidad concreta que queremos.

La simulación del comportamiento de una clase se hace mediante los “dobles” que pueden ser de distintos tipos:

- **Dummy**: Son objetos que se utilizan para realizar llamadas a otros métodos, pero no se usan.
- **Stub**: es como un dummy ya que sus métodos no hacen nada, pero devuelven cierto valor que necesitamos para ejecutar nuestro test con respecto a ciertas condiciones.
- **Spy**: Es un objeto real que permite verificar el uso que se hace del propio objeto, por ejemplo el número de veces que se ejecuta un método o los argumentos que se le pasan.
- **Mock**: Es un stub en el que sus métodos sí implementan un comportamiento, pues esperan recibir unos valores y en función de ellos devuelve una respuesta.
- **Fake**: Son objetos que tienen una implementación que funciona pero que no son apropiados para usar en producción (por ejemplo, una implementación de HttpSession).

Ejemplo práctico con JUnit 4 y Mockito

A continuación se muestra un ejemplo práctico con test unitarios y de integración entre dos componentes. Se trata de resolver una ecuación de primer grado del tipo: **ax + b = c**, donde a, b y c son números enteros. Por lo tanto la solución a dicha ecuación es **x = (c - b) / a**

En el código **"a"** lo llamamos **parte1**, **"b"** lo llamamos **parte2**, **"+"** o **"-"** es el **operador** y **"c"** es la **parte3**.

El proyecto tienes 2 clases: **EcuacionPrimerGrado** con el método que tiene la fórmula para resolver la ecuación y **Parseador**, que se encarga de realizar el parseo de la cadena con la ecuación.

```
package com.softtek.ecuacion;

/**
 * Ecuacion de primer grado
 * Solucion:  x = (c - b) / a
 * es decir:  x = (parte3 - parte2)/parte1
 */
public class EcuacionPrimerGrado {

    private Parseador parseador = new Parseador();

    public double obtenerResultado(final String ecuacion) {

        int parte1 = parseador.obtenerParte1(ecuacion);
        int parte2 = parseador.obtenerParte2(ecuacion);
        int parte3 = parseador.obtenerParte3(ecuacion);
        double resultado = Double.valueOf((parte3 - parte2)) / Double.valueOf(par
        return resultado;
    }

}
```

El **Parseador** tiene el siguiente código:

```
package com.softtek.ecuacion;

public class Parseador {

    public int obtenerParte1(final String ecuacion) {

        String[] partes1 = obtenerPartes12(ecuacion);

        String parte1 = partes1[0].trim();

        return Integer.valueOf(parte1.substring(0, parte1.length() - 1));
    }

    public int obtenerParte2(final String ecuacion) {

        String[] partes1 = obtenerPartes12(ecuacion);

        String parte2 = partes1[1].trim();

        String operador = obtenerOperador(ecuacion);

        if ("-".equals(operador)) {
            return Integer.valueOf(parte2) * (-1);
        }

        return Integer.valueOf(parte2);
    }

    public String obtenerOperador(final String ecuacion) {
        if (ecuacion.indexOf('+') > 0) {
            return "+";
        } else {
            return "-";
        }
    }

    public int obtenerParte3(final String ecuacion) {
        String[] partesEcuacion = ecuacion.split("=");
        return Integer.valueOf(partesEcuacion[1].trim());
    }

    private String[] obtenerPartes12(final String ecuacion) {
```

```
            String[] partesEcuacion = ecuacion.split("=");

            String operador = obtenerOperador(ecuacion);

            String[] partes1 = partesEcuacion[0].split("\\" + operador);

            return partes1;
        }
    }
```

A continuación se muestra los tests unitarios del parseador. El **SUT** es la clase Parseador y no tiene ninguna dependencia con ningún otro componente.

```
package com.softtek.ecuacion;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class ParseadorTest {

    private final Parseador parseador = new Parseador();

    @Test
    public void obtenerParte1Unidades() {

        String ecuacion1 = "2x - 1 = 0";

        int resultado = parseador.obtenerParte1(ecuacion1);

        assertEquals(2, resultado);
    }


    @Test
    public void obtenerParte2Suma() {

        String ecuacion1 = "2x + 1 = 0";

        int resultado = parseador.obtenerParte2(ecuacion1);

        assertEquals(1, resultado);
    }

    @Test
    public void obtenerParte3Positivo() {

        String ecuacion1 = "2x + 1 = 3";

        int resultado = parseador.obtenerParte3(ecuacion1);

        assertEquals(3, resultado);
    }

    @Test
```

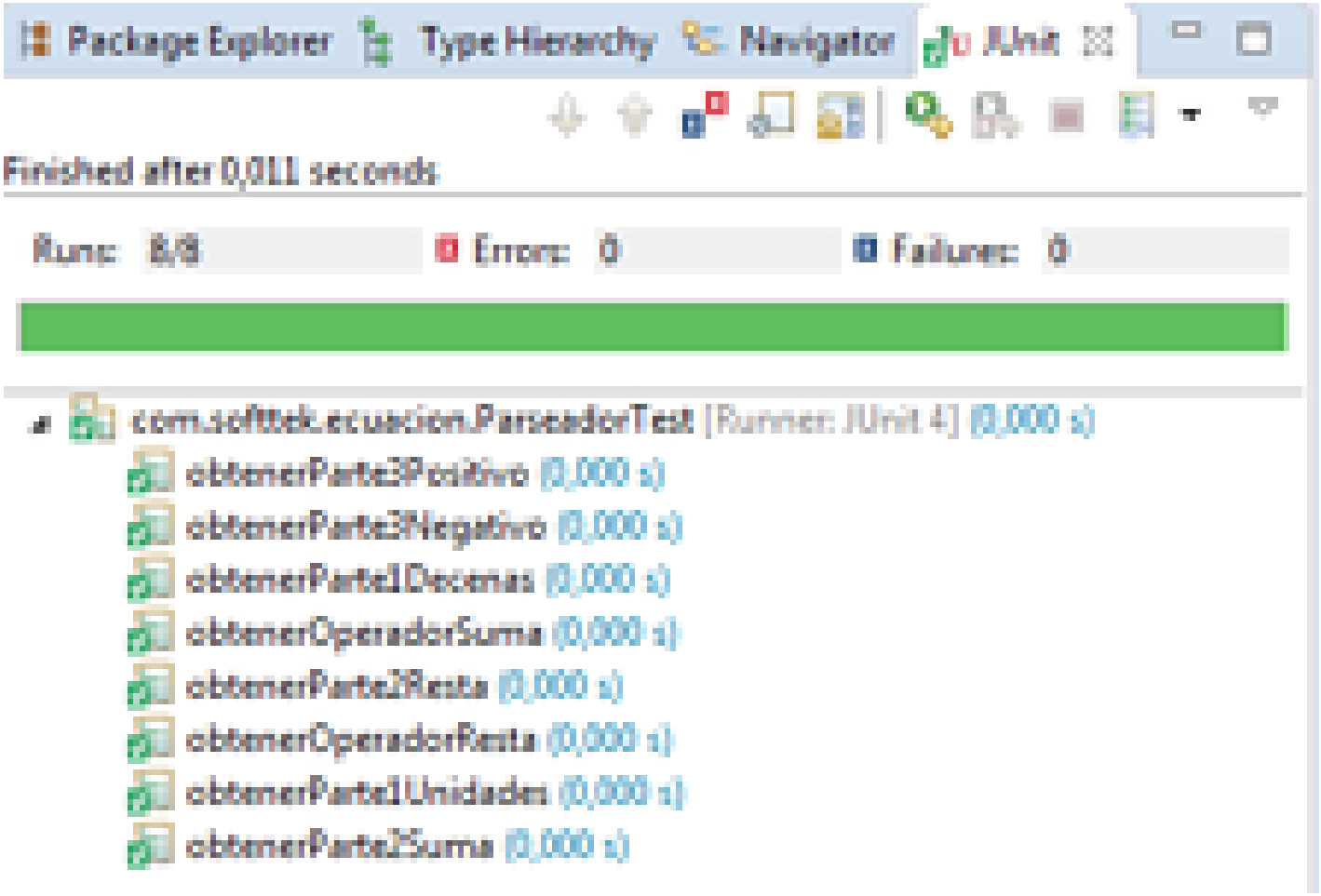
```
        public void obtenerOperadorSuma() {

            String ecuacion2 = "2x + 1 = 0";

            String operador = parseador.obtenerOperador(ecuacion2);

            assertEquals("+", operador);
        }
    }
```

El resultado de la ejecución de los tests se visualiza de la siguiente forma con Eclipse:



A continuación, se muestra un ejemplo de un test de integración donde se verifica la interacción del componente EcuacionPrimerGrado y el Parseador. Los test comprueban que el resultado final de la ecuación es correcto.

```java
package com.softtek.ecuacion;

import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class EcuacionPrimerGradoIntegrationTest {

    EcuacionPrimerGrado ecuacion = new EcuacionPrimerGrado();

    @Test
    public void solucionaEcuacionConMenos() {

        Double result = ecuacion.obtenerResultado("2x - 1 = 0");

        Double valueExpected = 0.5;

        assertEquals(valueExpected, result);
    }

    @Test
    public void solucionaEcuacionConMas() {

        Double result = ecuacion.obtenerResultado("2x + 1 = 0");

        Double valueExpected = -0.5;

        assertEquals(valueExpected, result);
    }

    @Test
    public void solucionaEcuacionConParte3Mayor0() {

        Double result = ecuacion.obtenerResultado("2x + 1 = 10");

        Double valueExpected = 4.5;

        assertEquals(valueExpected, result);
    }
```

```
    }
```

Ahora, si sólamente quisiéramos probar de forma unitaria el método **obtenerResultado** de la clase **EcuacionPrimerGrado** debemos hacer uso de los "***dobles***". Pensad en implementar estos dobles vosotros mismos sin hacer uso de ningún framework. Seguramente ya estáis pensando en crear una interfaz para la clase **Parseador** y crear métodos que permitan hacer la sustitución por un objeto **fake**...vamos, que se complica un poco.

Pues para esto podemos hacer uso de **Mockito**. Con **@InjectMocks**establecemos el objeto sobre el cual se realizará la inyección de los objetos marcados con **@Mock**, es necesario inicializar estos mocks con **MockitoAnnotations.initMocks(this)**; en un método de inicialización con **@Before**. Para establecer comportamientos del mock Parseador se utiliza **when**, antes de realizar la ejecución del test. A continuación podéis ver el ejemplo:

```java
package com.softtek.ecuacion;

import static org.junit.Assert.assertEquals;
import static org.mockito.Mockito.when;

import org.junit.Before;
import org.junit.Test;
import org.mockito.InjectMocks;
import org.mockito.Mock;
import org.mockito.MockitoAnnotations;

public class EcuacionPrimerGradoMockitoTest {

    @InjectMocks
    private EcuacionPrimerGrado ecuacionPrimerGrado;

    @Mock
    private Parseador parseador;

    @Before
    public void inicializaMocks() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void solucionaEcuacionConMenos() {
        String ecuacion = "2x - 1 = 0";

        when(parseador.obtenerParte1(ecuacion)).thenReturn(2);
        when(parseador.obtenerParte2(ecuacion)).thenReturn(-1);
        when(parseador.obtenerParte3(ecuacion)).thenReturn(0);

        Double result = ecuacionPrimerGrado.obtenerResultado(ecuacion);

        Double valueExpected = 0.5;

        assertEquals(valueExpected, result);
    }

    @Test
    public void solucionaEcuacionConMas() {

        String ecuacion = "2x + 1 = 0";
```

```
            when(parseador.obtenerParte1(ecuacion)).thenReturn(2);
            when(parseador.obtenerParte2(ecuacion)).thenReturn(1);
            when(parseador.obtenerParte3(ecuacion)).thenReturn(0);

            Double result = ecuacionPrimerGrado.obtenerResultado(ecuacion);

            Double valueExpected = -0.5;

            assertEquals(valueExpected, result);
        }
    }
```

Conclusión

Debemos asegurar la calidad de cualquier producto, también si es un producto de software. No debemos subestimar el plan de pruebas ni escatimar el tiempo que le dedicamos. Como profesionales debemos ofrecer un software robusto, libre de errores y que sea mantenible.

Todo el código de este post lo podéis encontrar mi perfil de github, en el proyecto practicaTesting.